SCHOOL OF **ELECTRICAL** AND
INFORMATION ENGINEERING
University of the Witwatersrand
Software Development II

Library Linking Guide

1 Introduction

A translation unit is the basic unit of compilation in C++. It consists of a `.cpp` source code file and all of the directly, and indirectly, included header files. The unit is translated into machine code which is stored in an object file (which has a `.obj` extension when using a Windows-based toolchain and a `.o` extension when using a Linux-based toolchain). The machine code in an object file is both [1]:

- *relocatable* — the relative memory addresses at which the functions will be placed is still undecided, and
- *unlinked* — external references to functions and global data have not yet been resolved.

Objects files may be linked into an executable by the linker, fully resolving all functions calls and relative addresses. Alternatively, they may be packaged as *libraries*. A library is simply an archive containing a number of object files [2, 3], similar to a zip archive, with the extension `.lib` (when using a Windows-based toolchain) or `.a` (when using a Linux-based toolchain). Creating a library is a way of packaging functionality so that it is easily accessible to clients, without requiring clients to compile the library code.

C++ libraries come in three forms:

- Header-only libraries (like `doctest`)
- Statically-linked libraries
- Dynamically-linked libraries (DLLs)

Statically-linked libraries and dynamically-linked libraries are also known as binaries as they are already compiled. When libraries are statically linked a copy of all of the library's functions is placed within the executable. This may result in a larger executable but it has advantages in that the executable is faster and more easily deployed, in that all of the library functionality is incorporated within a single file. Dynamically-linked libraries, on the other hand, package library functionality in a `.dll` file which can be shared amongst many different executables. Both the executable image, and the DLLs that it depends on, are loaded into memory at run-time and calls are made to functions contained within the DLL.

When creating your own executable and linking to compiled libraries, you are required to specify to the linker of the names of the library files that you will be linking with. If you are linking with a static library, the library file contains the compiled library functions which will be incorporated into your executable's file; if you are linking with a DLL, the library file contains the entry point information for the DLL, enabling the linker to resolve references to DLL functions called from within your executable.

In order to statically or dynamically link with libraries, the libraries need to have been built with the *same* compiler that is being used to compile the client code which uses the library.

2 Checking the Compiler Version — MinGW 7.3.0 (64-bit)

Before starting, check that you have the correct version of the compiler installed. To check the compiler version, find the installation binary directory (if you followed the installation guide it will be located at `c:\mingw64\bin\`) and view the contents. You should see a compiler executable called: `x86_64-w64-mingw32-gcc-7.3.0.exe`

Lastly, go to Settings|Build Settings...|Compilers in the CodeLite IDE and make sure that the path to the MinGW (mingw64) compiler is `C:/mingw64/bin/g++.exe --std=c++17`

3 Library Linking Procedure

These instructions consist of five basic steps:

1. creating a sample project using the test code,
2. including the library's header files (no further steps are required for header-only libraries),
3. linking with the library's binary files,
4. making the DLL files available in the case of dynamic linking,
5. building and running the sample project.

Note, the steps and procedures given for linking in libraries are specific to the Windows platform but they are similar on other platforms.

The following sections explain how to change the compiler/linker settings on a per-project basis. Note, that the include and linker paths can also be set globally from Settings|Build Settings…|Compilers|Advanced.

3.1 Create a Sample Project

Test code is provided for each of the libraries that are available on the "Laboratories" page of the course web site. Create a new project in CodeLite and add to it all the source code files found in the test code directory.

Note, when creating a project the project type should be "Simple executable (g++)". Select "MinGW (mingw64)" as the compiler. If you use a different compiler, or a different version of the MinGW compiler, *you will have linker errors* because the libraries have been built with this particular compiler.

3.2 Include the Library's Header Files

Every project that uses a library needs to include the header files for that library. To do this do the following:

1. Specify the search path for the header files.
   Right-click on the project name and select Settings…|Compiler and fill in the path to the **`.h`** files of the appropriate library at the "Include Paths" field.

For **doctest** this is: `<path to doctest folder>\doctest-1.2.9\doctest\`
For **SFML** this is: `<path to SFML folder>\SFML-2.5.0\include\`

2. Include the library headers in your source code, where necessary.
   Insert `#include` directives in your source files to include the header file/s (this has already been done for the test code). Note that the path and filename specified in the `#include` directive is appended to the search path specified in the previous step.

3.3 Link with the Library's Binary Files (SFML only)

The linker has to be setup to link with the library files as follows:

1. Specify the search path for the library files.
   In the project settings, select the Linker tab and supply the path to the **`lib`** directory containing the library files in the "Libraries Search Path" field.
2. Specify the actual library files that need to be linked by adding the name of each file to the "Libraries" field.

   For **SFML** input: `sfml-audio;sfml-graphics;sfml-window;sfml-system`

3.4 Make the DLL Files and Other Resources Accessible (SFML Only)

1. For dynamically-linked libraries the DLL files have to be available in *the same directory as the executable* for the application to use at run-time. The executable that is generated, when your project is built, is placed in the `Debug` directory located at `<path to codelite workspace>\<test code project name>\Debug`.

   The SFML library's DLL files are contained in the library's `bin` directory. Copy the files listed below from this directory into the `Debug` directory.
   (a) sfml-audio-2.dll
   (b) sfml-graphics-2.dll
   (c) sfml-window-2.dll
   (d) sfml-system-2.dll
   (e) openal32.dll
2. Copy any *resources required by your program* to the `Debug` directory as well (and make sure that your code has the correct path to these resources). The resources for the SFML test program can be found in the `resources` directory within the test code directory. Copy the entire `resources` directory into your `Debug` directory.

3.5 Build and Run the Sample Project

You are now in a position to build and run the sample project. Make sure the project is activated (project name appears in bold type) and press Control-F9 to build and run it.

3.5.1 Hide the Console Window

If you want to hide the console window for SFML projects right-click on the project name and select Settings…|General and tick the box that says "This program is a GUI application"

in the Execution section. Select Linker from the tree on the left and specify the Windows subsystem by inputting: `-Wl,--subsystem,windows` for the Linker Options.

References

[1] J. Gregory. *Game Engine Architecture*. A K Peters, 1st ed., 2009.

[2] Microsoft. "Microsoft Library (.LIB) Format, Created by LIB.EXE." http://support.microsoft.com/kb/79259, Last accessed: 10/08/2011.

[3] O. L. Astrachan. "Creating Unix Libraries." http://www.cs.duke.edu/~ola/courses/programming/libraries.html, Last accessed: 10/08/2011.